Dağıtık GPU'larda Hesaplama
Distributed Computation on GPUs

Öğrenci-1
Berşan Muhammet Ekici
İstanbul Ticaret Üniversitesi
Bilgisayar Mühendisliği
İstanbul, TR
brsnekici@hotmail.com

Öğrenci-2
Ahmet Burak Kara
İstanbul Medeniyet Üniversitesi
Elektrik Elektronik Mühendisliği
İstanbul, TR
abkkara@gmail.com

Akademik Danışman
Dr.Feyza Merve Hafızoğlu
İstanbul Ticaret Üniversitesi
Bilgisayar Mühendisliği
İstanbul, TR
Dr.Feyza Merve Hafızoğlu

Sanayi Danışmanı
Özhan Taşdemir
TUSAŞ – Türk Havacılık ve Uzay Sanayii A.Ş.
Ankara, Türkiye
ozhan.tasdemir1@tai.com.tr

***Özetçe*—Bu çalışma, TUSAŞ bünyesindeki çoklu GPU kaynaklarının dağıtık derin öğrenme amacıyla etkin ve senkronize biçimde kullanılabilmesini sağlamak üzere tasarlanmış bir eğitim altyapısını sunmaktadır. Sistem, socket tabanlı bir haberleşme protokolüyle farklı düğümlerdeki GPU'ların eşzamanlı çalışmasını sağlar. YOLOv8 modeli üzerinde yapılan uygulamada, parametre birleştirme işlemleri hem CPU hem de GPU üzerinde test edilmiş, GPU tabanlı parallell_update fonksiyonunun en yüksek verimi sağladığı gözlemlenmiştir. Gerçekleştirilen performans testleri ile eğitim sürelerinde %77'ye varan iyileşmeler kaydedilmiştir. Geliştirilen sistem, büyük ölçekli yapay zeka modellerinin daha kısa sürede eğitilmesini mümkün kılarken, TUSAŞ'ın donanım altyapısının daha verimli kullanılmasını amaçlamaktadır.**

***Anahtar Kelimeler* — dağıtık eğitim; GPU paralelleştirme; socket haberleşme; model ve performans senkronizasyonu**

***Abstract*—This study presents a distributed training architecture designed to utilize multiple GPU resources within TUSAŞ efficiently and in sync for deep learning tasks. The system enables synchronized execution across different nodes via a socket-based communication protocol. Applied on the YOLOv8 model, the architecture supports both CPU and GPU-based parameter aggregation, with the parallell_update GPU function demonstrating the highest efficiency. Experimental results show up to 77% reduction in epoch time. The system aims to accelerate the training of large-scale AI models while enhancing the utilization of TUSAŞ's computational infrastructure.**

Keywords — distributed training; GPU parallelism; socket communication; model synchronization; performance optimization

I. GİRİŞ

Günümüzde derin öğrenme uygulamalarının ölçeği hızla büyümekte ve beraberinde yüksek hesaplama gücüne duyulan ihtiyaç da artmaktadır. Bu ihtiyaç, özellikle bilgisayarla görü (computer vision) ve doğal dil işleme (NLP) gibi alanlarda, büyük veri kümeleri üzerinde gerçekleştirilen model eğitimlerinde daha da belirgin hâle gelmiştir. Bu tür eğitim süreçleri, çoğunlukla yüksek sayıda GPU kaynağı gerektirmektedir.[1]

Türk Havacılık ve Uzay Sanayii A.Ş. (TUSAŞ), bünyesinde yüzlerce GPU barındıran güçlü bir altyapıya sahiptir. Ancak bu GPU'lar genellikle birbirinden bağımsız çalışan farklı makinelerde yer almakta; bu da büyük ölçekli model eğitimlerinin verimli şekilde gerçekleştirilmesini zorlaştırmaktadır. Toplam GPU kapasitesi yüksek olmasına rağmen, bu kaynakların dağınık yapısı nedeniyle mevcut hesaplama potansiyeli tam anlamıyla kullanılamamaktadır.

Bu projede amaç, TUSAŞ'ın elindeki mevcut GPU kaynaklarını merkezi bir sistem altında birleştirerek, dağıtık bir eğitim altyapısı oluşturmaktır. Bu altyapı sayesinde, farklı makinelerde bulunan GPU'lar ortak bir görev üzerinde senkronize şekilde çalıştırılabilecek; model eğitimi, master-node kontrolünde güvenli ve koordine biçimde yürütülebilecektir. Böylece, büyük veri kümeleri ve karmaşık modeller için eğitim süresi azaltılırken, mevcut kaynakların atıl kalmasının önüne geçilmiş olunacaktır.

Geliştirilen sistem, dağıtık eğitim desteğiyle entegre şekilde çalışan YOLOv8 modeliyle test edilip başarıyla devreye alınmıştır. Mimari, benzer derin öğrenme modellerine veya herhangi bir vektörel hesaplayıcıya uyarlanabilecek esneklikte tasarlanmıştır.

II. YÖNTEM

Geliştirilen sistem, farklı fiziksel makinelerdeki GPU'ları merkezi bir yapı altında eş zamanlı şekilde çalıştırarak dağıtık bir model eğitimi gerçekleştirmeyi amaçlamaktadır. Mimari, temel olarak bir "master" ve birden fazla "node" (istemci)

yapısından oluşmaktadır. Master, model parametrelerinin birleşimini yaparken, her node kendi yerel verisi üzerinde eğitimi gerçekleştirmekte ve güncellenmiş ağırlıkları master'a göndermektedir.

A. Socket Tabanlı Ağ İletişimi:

Sistem, TCP/IP protokolü üzerinden özel olarak tasarlanmış bir socket tabanlı iletişim altyapısı ile çalışmaktadır. Master tarafında bir *DuplexServer* çalışmakta ve istemcilerden gelen model parametrelerini *getData()* fonksiyonu ile almaktadır. Her istemci *DuplexClient* sınıfı ile bağlanmakta, modelin güncel ağırlıklarını ve loss değerlerini pickle formatında master'a iletmektedir.

Alternatif olarak, sistem başlangıçta UNIX pipe (named pipe) tabanlı iletişim ile de test edilmiştir. Pipe yapıları küçük boyutlu kontrol mesajları veya düşük hacimli veri transferleri için düşük gecikme avantajı sunmaktadır. Ancak model ağırlıkları gibi büyük veri bloklarının aktarımında pipe sistemlerinin buffer sınırlamaları nedeniyle darboğaza yol açabildiği gözlemlenmiştir. Bu nedenle socket mimarisi, yüksek hacimli ve sık veri transferi gerektiren dağıtık eğitim yapıları için daha uygun ve kararlı bir çözüm sunmuştur.

Bu yapı sayesinde:

- Donanımdan bağımsız, esnek bir iletişim mimarisi kurulmuştur.
- Eğitim sırasında parametre paylaşımı gecikmesiz ve düşük yükle gerçekleşmektedir.

B. GPU Benchmarklama ve Dinamik Görev Dağılımı:

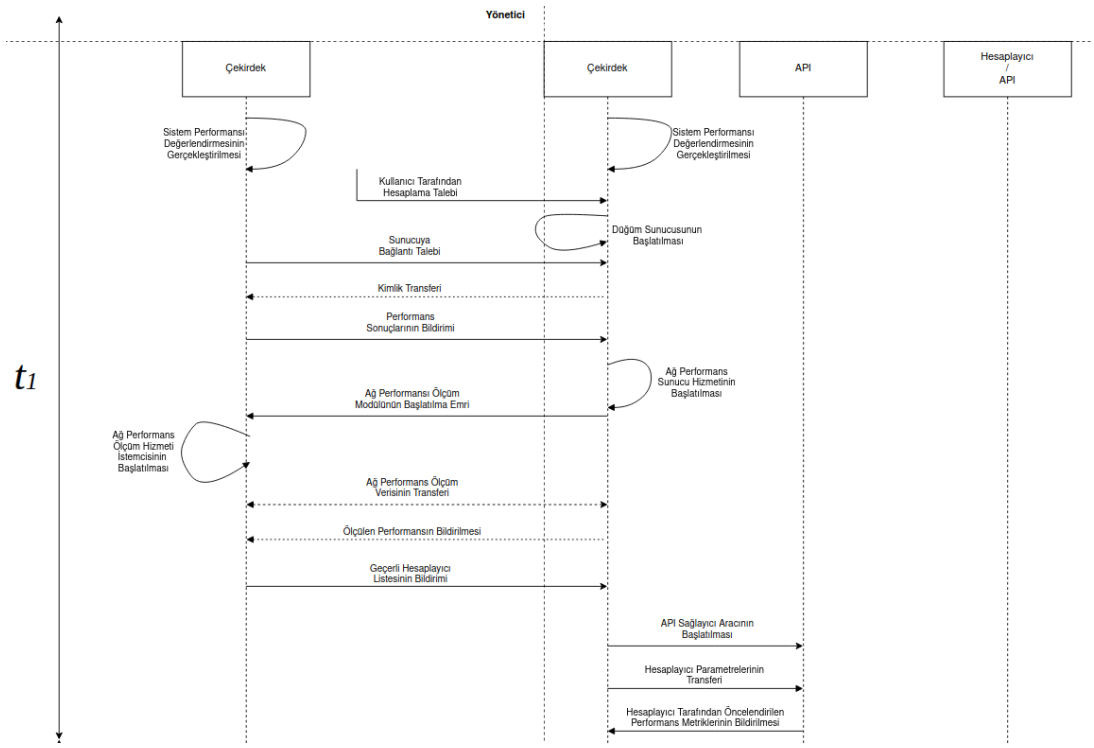
Eğitim başlamadan önce, her düğümdeki GPU'nun hesaplama gücünü değerlendirmek amacıyla *gpuBench.cu* dosyasında tanımlı CUDA tabanlı bir benchmark uygulanır. Bu benchmark, her bir GPU'nun FP16, FP32 ve FP64 işlemlerindeki performansını ölçerek sistem tarafından puanlanmasını sağlar. Bu puanlar *systemParser.cpp* aracılığıyla merkezi sistem tarafından okunur ve ağdaki her node'un uygun iş yüküyle eşleşmesini sağlamak için görev dağılımı bu skorlar dikkate alınarak yapılır. Bu sayede daha güçlü GPU'lar daha fazla veriyle çalışırken, zayıf GPU'lar daha az yük alır.

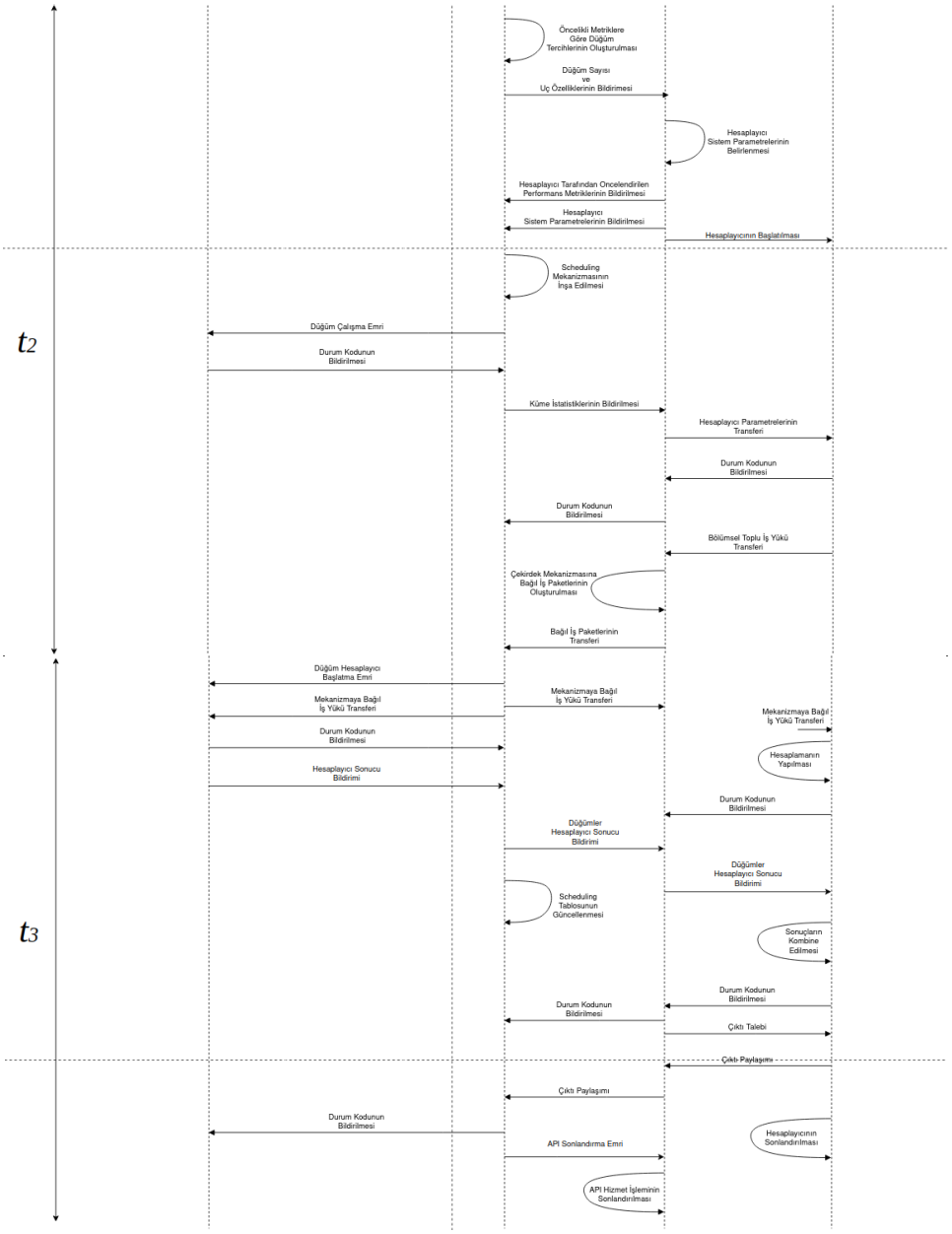
Aynı zamanda, bu GPU'ların bellek kapasiteleri de hesaba katılarak, her bir düğümün işlemekte zorlanmayacağı maksimum kernel boyutları önceden belirlenir. Böylece bellek taşmaları önlenerek eğitim sürecinin kesintisiz devam etmesi sağlanır. her düğümdeki GPU'nun hesaplama gücünü değerlendirmek amacıyla *gpuBench.cu* dosyasında tanımlı CUDA tabanlı bir benchmark uygulanır. Bu benchmark, her bir GPU'nun FP16, FP32 ve FP64 işlemlerindeki performansını ölçerek sistem tarafından puanlanmasını sağlar. Bu puanlar *systemParser.cpp* aracılığıyla merkezi sistem tarafından okunur ve ağdaki her node'un uygun iş yüküyle eşleşmesini sağlamak için görev dağılımı bu skorlar dikkate alınarak yapılır. Bu sayede daha güçlü GPU'lar daha fazla veriyle çalışırken, zayıf GPU'lar daha az yük alır.

C. Eğitim Süreci ve YOLOv8 Entegrasyonu:

Sistemin eğitim süreci, *BaseTrainer* sınıfı etrafında şekillenmiştir. Bu sınıf, YOLOv8 modelinin PyTorch üzerinden eğitilmesini sağlamaktadır. Her node kendi verisinde train_loader üzerinden batch'ler ile eğitim yapar. Eğitim sürecinde *elle_batchi_hallet()* fonksiyonu ile ileri ve geri yayılım yapılır, ardından oluşan model ağırlıkları pickle ile dışa aktarılır.

- Eğitim tamamlandığında, node güncellenmiş ağırlıklarını master'a gönderir.





- Master node, gelen ağırlıkları toplar ve *parallell_update()* fonksiyonu ile modelin ortak halini oluşturur.Parametre Güncelleme Yöntemleri:

1) CPU ile Thread Tabanlı Güncelleme:

thread_list(benkimim) fonksiyonu ile her parametre anahtarının eşlenmesi paralel thread'ler üzerinden yapılabilir. 6 thread ile parametre listesi mod 6'ya göre dağıtılarak eşzamanlı güncelleme gerçekleştirilir.

2) GPU ile In-Place Güncelleme

parallell_update(model, karsi_model_state, state_key_list, benkimim) fonksiyonu, PyTorch'un *add_()* ve *div_()* gibi in-place tensor işlemleriyle doğrudan GPU üzerinde çalışır. Bu, parametrelerin çok hızlı ve belleği kopyalamadan birleştirilmesini sağlar. *torch.no_grad()* bloğu içinde çalıştırılarak eğitimi etkilemeden güncelleme yapılır.

D. Senkronizasyon ve Hata Yönetimi:

Eğitim döngüsü, her epoch sonunda tüm node'ların parametrelerini toplayarak ilerlemektedir. Eğer bir node bağlantı kuramazsa ya da veri gönderemezse, sistem zaman aşımları ve fallback mekanizmaları ile bu durumu algılar. Gerekirse eğitim tekrar başlatılır veya ilgili node geçici olarak devre dışı bırakılır.

III. YOLOV8

Bu çalışmada YOLOv8 modeli, proje isterlerinde özellikle talep edilmesi nedeniyle dağıtık yapay zeka eğitim süreçlerinde test edilen ilk hedef model olarak kullanılmıştır. Ultralytics tarafından geliştirilen bu model, nesne algılama görevlerinde güçlü doğruluk oranları ve güncel Torch

mimarisi ile öne çıkmaktadır. Modelin mimarisi, sistemde *BaseTrainer* sınıfı aracılığıyla dağıtık eğitim sistemine entegre edilmiştir.

A. Eğitim Altyapısına Uyarlama:

Diyagramda da gösterildiği gibi, her bir node yerel verisiyle eğitimi tamamladıktan sonra, model ağırlıklarını *state_dict* formatında master node'a socket üzerinden iletir. Master node, bu verileri toplayarak GPU üzerinden parametreleri birleştirir ve güncellenmiş ortak modeli tekrar düğümlere dağıtır. Bu döngü, her epoch sonunda tekrarlanır ve modelin ortaklaşa öğrenmesini sağlar.

Master node üzerinde gelen tüm ağırlıklar iki farklı yöntemle birleştirilebilmektedir:

- İlk aşamada geliştirilen *thread_list(benkimim)* fonksiyonu, CPU üzerinde çok iş parçacıklı (multi-threaded) güncelleme yapmaktadır. Parametre isimleri *state_key_list* üzerinden alınır ve bu liste mod 6'ya göre alt parçalara ayrılarak 6 ayrı thread'e bölüştürülür. Her thread, kendine ait parametreler için karşı modelden alınan değerleri güncelleyip ortalamaya dahil eder. Bu yöntem, erken prototipleme aşamasında işlevsel olmuş ancak özellikle büyük modellerde performans açısından yetersiz kalmıştır.
- Bu gözlemler üzerine geliştirilen *parallell_update()* fonksiyonu ise model parametrelerini doğrudan GPU üzerinde PyTorch'un in-place tensor işlemleri *(add_() ve div_())* ile birleştirir. Bellek verimliliği sağlayan bu yöntem, işlem süresini ciddi ölçüde azaltmış ve dağıtık sistemin toplam eğitim süresinde belirgin kazanımlar elde edilmesini mümkün kılmıştır. Tüm düğümlerden gelen parametreler toplanır, kümülatif olarak *add_()* ile birleştirilir ve düğüm sayısına bölünerek ortalama model oluşturulur. İşlem *torch.no_grad()* bloğu içinde gerçekleştirilir.

Sonuç olarak, thread tabanlı çözüm sistemin ilk sürümünde önemli bir rol oynamış, ancak paralel GPU işlemine geçiş performans açısından kalıcı çözüm sunmuştur. Her iki yöntem de senkronizasyon açısından güvenli olacak şekilde tasarlanmış ve eğitim döngüsünün sonunda merkezi modelin kararlı şekilde güncellenmesini sağlamıştır.

B. Veri Dağılımı ve Uyum

YOLOv8 modelinde başarımı etkileyen önemli faktörlerden biri veri dağılımıdır. Eğitim seti tüm düğümlere dengeli ve temsili biçimde dağıtılarak her bir node'un anlamlı katkı sağlaması hedeflenmiştir. Bu sayede modelin aşırı öğrenmesi veya eksik temsili engellenmiştir.

C. Performans Gözlemleri:

Dağıtık eğitim yapısı, YOLOv8'in tek bir GPU üzerinde eğitilmesine göre daha kısa sürede convergence sağlamış; zaman kazanımı ve kaynak verimliliği açısından ciddi avantajlar sunmuştur. Eğitim çıktıları, senkronizasyon yapısının tutarlı çalıştığını ve performansın merkezi olmayan yapıya rağmen yüksek kaldığını göstermektedir.

IV. ÇEKİRDEK

Sistemin çekirdeği, C++ ve CUDA tabanlı modüllerden oluşmaktadır. Eğitim sürecinin yönetimi, veri yapılarının oluşturulması, düğümler arası koordinasyon ve GPU'ların verimli biçimde kullanılması bu çekirdek mimari üzerinden yürütülmektedir.

A. Sistem Yapılandırması:

systemParser.cpp ve *systemParser.h* dosyaları, sistemdeki donanım ve düğüm yapılandırmalarını tanımlamak için kullanılmaktadır. Bu modül, her bir GPU'nun özelliklerini, bağlı node'ların IP adreslerini ve benchmark sonuçlarını içeren yapılandırma dosyasını işler. Bu bilgiler JSON formatında tanımlanmakta ve *systemParser* sınıfı tarafından ayrıştırılmaktadır. Her GPU için benchmark sonuçları *gpuBench* çıktılarından alınır ve hangi node'a ne kadar iş yükü düşeceği burada belirlenir.

B. GPU Performans Ölçümü:

CUDA tabanlı bu modül, her bir GPU'nun FP16, FP32 ve FP64 işlemlerindeki ham işlem performansını ölçmektedir. *gpuBench()* fonksiyonu, her işlem tipi için belirli bir sürede yapılan işlem sayısını hesaplar ve bu değerleri karşılaştırılabilir skorlar haline getirir. Bu skorlar sistemParser aracılığıyla sistemin geri kalanına iletilir ve iş yükü dağılımı bu bilgilere göre yapılır. Aynı zamanda GPU bellek kapasitesi de burada analiz edilerek kernel boyutlarının sınırları belirlenir.

C. Küme İşlemleri:

Diyagramda da gösterildiği gibi, sistem başlatıldığında her bir node, öncelikle kendi GPU'sunda *gpuBench* fonksiyonunu çalıştırarak performansını ölçer. Bu bilgiler master node'a socket aracılığıyla iletilir. Master bu verileri aldıktan sonra, eğitim sürecini başlatır ve her node'a iş yükü atar. Eğitim tamamlandığında, tüm düğümler güncellenmiş *state_dict* yapısını yeniden master'a göndererek modelin ortak biçimde güncellenmesini sağlar.

D. Ağ Arayüzü:

networkInterface.cpp ve *networkInterface.h* dosyaları, TCP/IP tabanlı ağ haberleşmesini yöneten yapıdır. Her bir node'un diğerleriyle veri alışverişi bu arayüz üzerinden sağlanır. Veriler socket üzerinden gönderilip alınırken hata yönetimi, yeniden bağlanma ve timeout işlemleri burada tanımlıdır. Bu yapı, Python tarafındaki *DuplexClient* ve *DuplexServer* yapılarına denk düşer.

V. ÇEKİRDEK ENTEGRASYONU

Bu bölümde Python ile yazılmış eğitim kodlarının C++/CUDA çekirdek ile nasıl entegre edildiği, socket üzerinden veri aktarımı, model parametrelerinin alınması ve GPU üzerinde nasıl işlendiği anlatılacaktır.

A. Paket Decoding:

Eğitim sırasında her bir düğüm, kendi yerel modelini eğitirken belirli aralıklarla diğer düğümlerden güncel parametreleri alır. Bu işlem, *DuplexServer* üzerinden alınan pickled paketlerin *getData()* fonksiyonuyla çözülmesiyle başlar. Bu aşamada:

- Her node, socket üzerinden gelen veriyi *getData()['data']* ifadesiyle ayrıştırır.
- Gelen parametreler *cPickle.loads()* ile tensor formatına geri çevrilir.
- *karsi_model* adı verilen geçici yapıya atanır.

Bu adımın temel amacı, diğer node'ların elde ettiği ara model çıktılarının elde edilmesi ve güncel modelle birleştirilmesine zemin hazırlamaktır.

B. Düğüm Sonuçlarının Kombine Edilmesi:

Model parametrelerinin birleştirilmesi, ilk olarak *thread_list()* fonksiyonu ile çok iş parçacıklı (multi-threaded) olarak tasarlanmıştır. Bu fonksiyon, parametre listesini altışarlı parçalara ayırarak farklı thread'lere dağıtır. Ancak bu yapı, zamanla yerini GPU ile yapılan daha hızlı ve verimli *parallell_update()* fonksiyonuna bırakmıştır.

thread_list(), CPU üzerinden her bir *state_dict* parametresine sırayla erişerek diğer düğümden gelen değerle toplar ve gerekirse ortalamasını alır. Ancak bu işlem, CPU sınırlamaları nedeniyle verimsizdir.

parallell_update() fonksiyonu ise *with torch.no_grad* bloğu altında doğrudan GPU üzerinde çalışarak:

- Parametreleri in-place *(add_)* olarak toplar.
- "tracked" ifadesi içermeyen tensor'lar için *div_* ile ortalama alır.
- CUDA backend'i üzerinden daha hızlı çalışır ve *thread_list()* fonksiyonuna göre belirgin performans kazanımı sağlar.

Bu süreçte önemli bir faktör de kernel boyutlarının her bir GPU'nun bellek kapasitesine göre sınırlandırılmasıdır. *gpuBench.cu* dosyası üzerinden her bir GPU'nun maksimum kapasitesi belirlenerek hangi büyüklükte modelleri işleyebileceği önden tespit edilir ve dağıtım bu doğrultuda şekillenir.

VI. Elde Edilen Sonuçlar ve Değerlendirme

Farklı yöntemler sonucu elde ettiğimiz eğitim performans metrikleri, her bir epoch için şu şekildedir:

- Veri transferi süresi: 0.8 saniye
- Model verileri yükleme süresi: 1.4 saniye
- Epoch süresi (Tek thread): 10 saniye
- Epoch süresi (Çift thread): 7.82 saniye
- Epoch süresi (GPU, Torch grad): 6.52 saniye

Verilere göre; iki bilgisayar kullanan sistemlerde ideal epoch süresi, tek thread'li sistemin yarısı olan 5 saniye olarak hesaplanmaktadır. Bu hesaba göre, GPU kullanan *parallell_update()* fonksiyonu ile elde edilen 6.52 saniye, ideal değerin yaklaşık %77'sine karşılık gelmektedir.

Bu sonuç, dağıtık eğitimin doğru şekilde çalıştığını ve GPU entegrasyonunun sistemde anlamlı bir hızlanma sağladığını göstermektedir. İlerleyen çalışmalarda daha fazla düğüm, gelişmiş veri paylaşım protokolleri ve daha güçlü GPU'lar ile bu oranın daha da iyileştirilmesi mümkündür.

VII. Çalışmanın Tusaş'a Olan Olasi Katkisi

Bu çalışma, TUSAŞ'ın sahip olduğu yüksek GPU donanım altyapısını daha verimli kullanabilmesini hedeflemektedir. Dağıtık yapay zeka eğitimi için geliştirilen mimari, kurum içinde bulunan çok sayıda güçlü GPU'nun birbirine bağlı sistemler hâlinde organize çalışmasını mümkün kılmaktadır. Böylece:

- Eğitim süreleri önemli ölçüde azaltılabilir; büyük ölçekli modeller günler yerine saatler içinde eğitilebilir.
- GPU kaynaklarının performansına göre dinamik iş yükü dağıtımı sağlanarak enerji ve zaman verimliliği artırılır.
- Merkezi olmayan eğitim yapısı sayesinde farklı birimlerdeki kaynaklar ortak görevlerde kullanılabilir hâle gelir.
- Ar-Ge süreçlerinde daha fazla model ve senaryo test edilebilir; deneme süresi kısalır, modelleme verimliliği artar.
- Açık kaynak tabanlı ve esnek mimari, mevcut yazılım sistemlerine kolaylıkla entegre edilebilir.

Bu sistem, yalnızca mevcut GPU altyapısının etkin kullanımı açısından değil, aynı zamanda yapay zekâ temelli sistemlerin sürdürülebilir ve ölçeklenebilir biçimde geliştirilmesi açısından da TUSAŞ'a stratejik avantajlar sağlayacaktır.

VIII. Öneriler, Alinan Dersler ve Gelecek Çalışmalar

CPU tabanlı çok iş parçacıklı (threaded) güncellemeler yüksek model boyutlarında performans engeli oluşturmaktadır; GPU üzerinde in-place güncelleme çok daha etkilidir.

Eğitim döngüsünde yaşanan küçük zaman kayıpları (örneğin pickle işlemi, veri transferi) toplam sürede önemli farklar yaratabilmektedir.

Basit socket mimarileri ile bile çok sayıda düğüm arasında başarılı senkronizasyon mümkündür.

IX. Kaynaklar

[1] N. Thompson, K. Greenewald, K. Lee, and G. Manso, "The Computatİonal Limİts of Deep Learnİng," 2020. [Çevrİmİçİ]. Erİşİm: https://İde.mİt.edu/wp-content/uploads/2020